Differential Flatness

Richard M. Murray, 13 Nov 2021 (updated 27 Nov 2022)

This notebook contains an example of using differential flatness as a mechanism for trajectory generation for a nonlinear control system. A differentially flat system is defined by creating an object using the `FlatSystem` class, which has member functions for mapping the system state and input into and out of flat coordinates. The `point_to_point()` function can be used to create a trajectory between two endpoints, written in terms of a set of basis functions defined using the `BasisFamily` class. The resulting trajectory is return as a `SystemTrajectory` object and can be evaluated using the `eval()` member function.

Example: bicycle model

To illustrate the methods of generating trajectories using differential flatness, we make use of a simple model for a vehicle navigating in the plane, known as the "bicycle model". The kinematics of this vehicle can be written in terms of the contact point \((x, y)\) and the angle \(\theta\) of the vehicle with respect to the horizontal axis:

\[
\begin{align*}
\dot{x} &= \cos \theta \, v \\
\dot{y} &= \sin \theta \, v \\
\dot{\theta} &= \frac{v}{l} \tan \delta
\end{align*}
\]

The input \(v\) represents the velocity of the vehicle and the input \(\delta\) represents the turning rate. The parameter \(l\) is the wheelbase.

We will generate trajectories for this system that correspond to a "lane change", in which we travel longitudinally at a fixed speed for approximately 40 meters, while moving from the right to the left by a distance of 4 meters.
It will be convenient to define a function that we will use to plot the results in a uniform way. In addition to the subplot, we also change the size of the figure to make the figure wider.

```python
In [2]:
    # Plot the trajectory in xy coordinates
    def plot_motion(t, x, ud):
        # Set the size of the figure
        # plt.figure(figsize=(10, 6))

        # Top plot: xy trajectory
        plt.subplot(2, 1, 1)
        plt.plot(x[0], x[1])
        plt.xlabel('x [m]')
        plt.ylabel('y [m]')
        plt.axis([x[0][0], x[0][1], x[0][0]-1, x[1][1]])

        # Time traces of the state and input
        plt.subplot(2, 4, 5)
        plt.plot(t, x[1])
        plt.ylabel('y [m]')

        plt.subplot(2, 4, 6)
        plt.plot(t, x[2])
        plt.ylabel('theta [rad]')

        plt.subplot(2, 4, 7)
        plt.plot(t, ud[0])
        plt.xlabel('Time t [sec]')
        plt.ylabel('v [m/s]')
        plt.axis([0, Tf, u0[0]-1, uf[0]+1])

        plt.subplot(2, 4, 8)
        plt.plot(t, ud[1])
        plt.xlabel('Time t [sec]')
        plt.ylabel('$\delta$ [rad]')
        plt.tight_layout()
```

Flat system mappings

To define a flat system, we have to define the functions that take the state and compute the flat “flag” (flat outputs and their derivatives) and that take the flat flag and return the state and input.

The `forward()` method computes the flat flag given a state and input:

```
zflag = sys.forward(x, u)
```

The `reverse()` method computes the state and input given the flat flag:

```
x, u = sys.reverse(zflag)
```
The flag \( \bar{z} \) is implemented as a list of flat outputs \( z_i \) and their derivatives up to order \( q_i \):

\[
zflag[i][j] = z_i^{(j)}
\]

The number of flat outputs must match the number of system inputs.

In addition, a flat system is an input/output system and so we define and update function \( (f(x, u)) \) and output (use `None` to get the full state).

```python
In [3]:
# Function to take states, inputs and return the flat flag
def bicycle_flat_forward(x, u, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a list of arrays to store the flat output and its derivatives
    zflag = [np.zeros(3), np.zeros(3)]

    # Flat output is the x, y position of the rear wheels
    zflag[0][0] = x[0]
    zflag[1][0] = x[1]

    # First derivatives of the flat output
    zflag[0][1] = u[0] * np.cos(x[2])  # dx/dt
    zflag[1][1] = u[0] * np.sin(x[2])  # dy/dt

    # First derivative of the angle
    thdot = (u[0]/b) * np.tan(u[1])

    # Second derivatives of the flat output (setting vdot = 0)
    zflag[0][2] = -u[0] * thdot * np.sin(x[2])
    zflag[1][2] = u[0] * thdot * np.cos(x[2])

    return zflag

# Function to take the flat flag and return states, inputs
def bicycle_flat_reverse(zflag, params={}):
    # Get the parameter values
    b = params.get('wheelbase', 3.)

    # Create a vector to store the state and inputs
    x = np.zeros(3)
    u = np.zeros(2)

    # Given the flat variables, solve for the state
    x[0] = zflag[0][0]  # x position
    x[1] = zflag[1][0]  # y position
    x[2] = np.arctan2(zflag[1][1], zflag[0][1])  # tan(theta) = ydot/xdot

    # And next solve for the inputs
    u[0] = zflag[0][1] * np.cos(x[2]) + zflag[1][1] * np.sin(x[2])
    thdot_v = zflag[1][2] * np.cos(x[2]) - zflag[0][2] * np.sin(x[2])
    u[1] = np.arctan2(thdot_v, u[0]**2 / b)

    return x, u
```
# Function to compute the RHS of the system dynamics

def bicycle_update(t, x, u, params):
    b = params.get('wheelbase', 3.)  # get parameter values
    dx = np.array([
        np.cos(x[2]) * u[0],
        np.sin(x[2]) * u[0],
        (u[0]/b) * np.tan(u[1])
    ])
    return dx

# Return the entire state as output (instead of default flat outputs)
def bicycle_output(t, x, u, params):
    return x

# Create differentially flat input/output system
bicycle_flat = fs.FlatSystem(
    bicycle_flat_forward, bicycle_flat_reverse,
    bicycle_update, bicycle_output,
    inputs=('v', 'delta'), outputs=('x', 'y', 'theta'),
    states=('x', 'y', 'theta'), name='bicycle_model')

print(bicycle_flat)

# Define a set of basis functions to use for the trajectories
polybasis = control.flatsys.PolyFamily(N)

# Plot out the basis functions
for k in range(poly.N):
    plt.plot(t, poly(k, t), label=f'k = {k}')
A BasisFamily object is used to construct trajectories for a flat system. The class must implement a single function that computes the jth derivative of the ith basis function at a time t:

\[ z_i^{(q)}(t) = \text{basis.eval_deriv}(self, i, j, t) \]

A basis set can either consist of a single variable that is used for each flat output (nvars = None) or a different variable for different flat outputs (nvars > 0).

**Attributes**

---

N : int
Order of the basis set.

**Polynomial basis functions.**

This class represents the family of polynomials of the form

.. math::
   \phi_i(t) = \left( \frac{t}{T} \right)^i

**Parameters**

---

N : int
Degree of the Bezier curve.

T : float
Final time (used for rescaling).
Approach 1: point to point solution, no cost or constraints

Once the system and basis function have been defined, the `point_to_point()` function can be used to compute a trajectory between initial and final states and inputs:

```python
traj = control.flatsys.point_to_point(sys, Tf, x0, u0, xf, uf, basis=polybasis)
```

The returned object has class `SystemTrajectory` and can be used to compute the state and input trajectory between the initial and final condition:

```python
xd, ud = traj.eval(timepts)
```

where `timepts` is a list of times on which the trajectory should be evaluated (e.g., `timepts = numpy.linspace(0, Tf, M)`).

In [5]:
```python
# Define the endpoints of the trajectory
x0 = np.array([0., -2., 0.]); u0 = np.array([10., 0.])
xf = np.array([40., 2., 0.]); uf = np.array([10., 0.])
Tf = 4

# Generate a normalized set of basis functions
poly = fs.PolyFamily(6, Tf)
```
# Find a trajectory between the initial condition and the final condition
traj = fs.point_to_point(bicycle_flat, Tf, x0, u0, xf, uf, basis=poly)

# Create the desired trajectory between the initial and final condition
timepts = np.linspace(0, Tf, 500)
xd, ud = traj.eval(timepts)

# Simulation the open system dynamics with the full input
t, y, x = ct.input_output_response(
    bicycle_flat, timepts, ud, x0, return_x=True)

# Plot the open loop system dynamics
plt.figure(1)
plt.suptitle("Open loop trajectory for unicycle lane change")
plot_motion(t, x, ud)

# Make sure the initial and final points are correct
print("x[0] = ", xd[:, 0])
print("x[T] = ", xd[:, -1])

x[0] = [-5.90673952e-15 -2.00000000e+00 -4.06619183e-16]
x[T] = [4.00000000e+01 2.00000000e+00 7.10542736e-16]

A look inside the code

The code to solve this problem is inside the file flatsys.py in the python-control package.
Here is what operative code inside the point_to_point() looks like:
# Map the initial and final conditions to flat output conditions
# We need to compute the output "flag": [z(t), z'(t), z''(t), ...]
# and then evaluate this at the initial and final condition.
#
zflag_T0 = sys.forward(x0, u0)
zflag_Tf = sys.forward(xf, uf)

# Compute the matrix constraints for initial and final conditions
# This computation depends on the basis function we are using. It
# essentially amounts to evaluating the basis functions and their
# derivatives at the initial and final conditions.

M_T0 = _basis_flag_matrix(sys, basis, zflag_T0, T0)
M_Tf = _basis_flag_matrix(sys, basis, zflag_Tf, Tf)

# Stack the initial and final matrix/flag for the point to point problem
M = np.vstack([M_T0, M_Tf])
Z = np.hstack([np.hstack(zflag_T0), np.hstack(zflag_Tf)])

# Solve for the coefficients of the flat outputs
# At this point, we need to solve the equation M alpha = zflag, where M
# is the matrix constrains for initial and final conditions and zflag =
# [zflag_T0; zflag_Tf].
#
# If there are no constraints, then we just need to solve a linear
# system of equations => use least squares. Otherwise, we have a
# nonlinear optimal control problem with equality constraints
# => use
# scipy.optimize.minimize().
#
# Start by solving the least squares problem
alpha, residuals, rank, s = np.linalg.lstsq(M, Z, rcond=None)
Approach #2: add cost function to make lane change quicker

```python
In [6]:
# Define timepoints for evaluation plus basis function to use
timepts = np.linspace(0, Tf, 20)
basis = fs.PolyFamily(12, Tf)

# Define the cost function (penalize lateral error and steering)
traj_cost = opt.quadratic_cost(
    bicycle_flat, np.diag([0, 0.1, 0]), np.diag([0.1, 1]), x0=xf, u0=uf)

# Solve for an optimal solution
start_time = time.process_time()
traj = fs.point_to_point(
    bicycle_flat, timepts, x0, u0, xf, uf, cost=traj_cost, basis=basis,
)
print("* Total time = %5g seconds\n" % (time.process_time() - start_time))

xd, ud = traj.eval(timepts)

plt.figure(2)
plt.suptitle("Lane change with lateral error + steering penalties")
plot_motion(timepts, xd, ud);
```

* Total time = 0.532655 seconds

Note that the solution has a very large steering angle (0.2 rad = ~12 degrees).
Approach #3: optimal cost with trajectory constraints

To get a smaller steering angle, we add constraints on the inputs.

```
In [7]:
    constraints = [
        opt.input_range_constraint(bicycle_flat, [8, -0.1], [12, 0.1])
    ]
    
    # Solve for an optimal solution
    traj = fs.point_to_point(
        bicycle_flat, timepts, x0, u0, xf, uf, cost=traj_cost,
        trajectory_constraints=constraints, basis=basis,
    )
    xd, ud = traj.eval(timepts)

    plt.figure(3)
    plt.suptitle("Lane change with penalty + steering constraints")
    plot_motion(timepts, xd, ud)
```

![Lane change with penalty + steering constraints](image)

**Ideas to explore**

- Change the number of basis functions
- Change the number of time points
- Change the type of basis functions: BezierFamily

```
In [8]:
    # Define a set of basis functions to use for the trajectories
    poly = fs.BezierFamily(6, 2)
```
# Plot out the basis functions

t = np.linspace(0, 2)
for k in range(poly.N):
    plt.plot(t, poly(k, t), label=f'k = {k}')

plt.legend()
plt.title("Bezier basis functions")
plt.xlabel("Time \(t\)")
plt.ylabel("\(\psi_i(t)\)");

In [ ]: